Algoritmos y Estructura de Datos: Examen 1 (Solución)

Grados Ing. Inf. y Mat. Inf. Noviembre 2011

Departamento de Lenguajes, Sistemas Informáticos e Ingeniería de Software

Apellidos: .....

Nombre: .....

DNI / NIE: ..... **Núm. matrícula:** .....

Normas

- Este examen consta de **5 preguntas** en **7 páginas**.
- La puntuación total del examen es de **10 puntos**.
- La duración total del examen es de **90 minutos**.
- El examen debe contestarse **en las hojas que se proporcionan**.
- Deben rellenarse los campos obligatorios **apellidos, nombre, y DNI/NIE**.
- Las calificaciones provisionales de este examen se publicarán en el Aula Virtual el **jueves 29 de noviembre de 2012** junto con las soluciones. La fecha de la revisión de este examen es el **lunes 3 de diciembre de 2012**. La hora y lugar de dicha revisión se anunciará en el Aula Virtual.
- En las preguntas con varias opciones, **sólo hay una respuesta válida por pregunta**. En este caso toda pregunta en que se marque más de una respuesta se considerará incorrectamente contestada y toda pregunta incorrectamente contestada restará del examen una cantidad de puntos $\frac{P}{O-1}$, donde P es la puntuación de la pregunta y O el número de opciones ofrecidas. Por ejemplo, una respuesta incorrecta en una pregunta de un 1 punto con 4 opciones resta $\frac{1}{3}$.

(1 punto) 1. El siguiente método toma como parámetro una lista de objetos de clase `Alumno` e indica si todos ellos aparecen en una lista de alumnos matriculados `listaMatriculados` cuyo tamaño es fijo (dicha lista no puede modificarse):

```
public bool matriculados(PositionList<Alumno> lista) {
  Iterator<Alumno> it = lista.iterator();
  boolean all = true;
  while (it.hasNext() && all)
    all = all && listaMatriculados.member(it.next());
  return all;
}
```

Se pide: Indicar de entre las siguientes posibilidades la complejidad en caso peor del método `matriculados`, donde n el tamaño de `lista`:

(a) $O(1)$ (b) $O(n^2)$ (c) $O(n)$ (d) $O(n \times m)$, con m el tamaño de `listaMatriculados`.

Esta pregunta es ambigua y se considera nula. La complejidad de `listaMatriculados.member(it.next())` es $O(1)$ pues el tamaño m de `listaMatriculados` es una constante. Si $n \leq m$ entonces n está acotado por una constante y por tanto la complejidad de `matriculados` es $O(1)$. Pero debemos considerar el caso peor $n > m$. El bucle `while` termina cuando se ha recorrido `lista` o al encontrar un alumno de `lista` que no está en `listaMatriculados`. Si todos los alumnos en `lista` están en `listaMatriculados` entonces para que $n > m$ se cumpla tiene que haber alumnos repetidos en `lista`. Si hay alumnos repetidos entonces la complejidad es $O(n)$. Si no los hay, entonces $n \leq m$ y la complejidad es $O(1)$ porque no puede cumplirse $n > m$ y que todos los alumnos de `lista` estén en `listaMatriculados` sin haber alumnos repetidos. Es irrelevante si `listaMatriculados` tiene alumnos repetidos. El caso peor es aquel en el que hay alumnos repetidos y la complejidad es $O(n)$. Sin embargo el enunciado no dice nada sobre alumnos repetidos y podría suponerse que $n \leq m$ es el caso en el contexto de la pregunta.

(2 puntos) 2. **Se pide:** Implementar en Java el método:

```
public void removeAll(PositionList<E> list, E e)
```

Dicho método debe eliminar todas las ocurrencias del objeto elemento `e` de la lista `list`. La lista debe quedar intacta si no hay ninguna ocurrencia del elemento.

Solución que usa el `remove` del iterador:

```
public void removeAll(PositionList<E> list, E e) {
  Iterator<E> it = list.iterator();
  while (it.hasNext())
    if (it.next().equals(e)) it.remove();
}
```

Solución que usa un «snapshot»:

```
public void removeAll(PositionList<E> list, E e) {
  for (Position<E> p : list.positions())
    if (p.element().equals(e)) list.remove(p);
}
```

(2 puntos) 3. Se dispone de un TAD para bases de datos definido por el interfaz `BaseDatos`. El interfaz ofrece el método `size()` que debe devolver el tamaño de un objeto base de datos.

Se dispone de una clase `BaseDatosRel` que implementa una base de datos relacional. Mostramos las declaraciones, omitiendo los contenidos del interfaz y la clase:

```
public interface BaseDatos { ... }
public class BaseDatosRel implements BaseDatos { ... }
```

Surge la necesidad de poder comparar objetos de tipo `BaseDatos`. El orden total consistirá en comparar según el tamaño. **Se pide:** Indicar cuál de entre las siguientes posibilidades describe la solución más adecuada en términos de corrección y reusabilidad. Los puntos suspensivos indican líneas de código omitidas en las que se crean los objetos bases de datos y se les insertan datos:

(a) Comparar los tamaños usando los operadores de comparación de Java:

```
public static void main(String [] s) {
  BaseDatos b1, b2;
  ...
  if (b1 < b2) System.out.println("b2 más grande");
}
```

(b) Usar el método `compareTo`:

```
public static void main(String [] s) {
  BaseDatos b1, b2;
  ...
  if (b1.compareTo(b2) < 0) System.out.println("b2 más grande");
}
```

(c) Definir y usar un comparador para `BaseDatos`: ✔

```
public class CompBD implements Comparator<BaseDatos> {
  int compare(BaseDatos b1, BaseDatos b2) {
    if (b1.size() < b2.size()) return -1;
    else if (b1.size() == b2.size()) return 0;
         else return 1;
  }
}
public static void main(String [] s) {
  BaseDatos b1, b2;
  CompBD c = new CompBD();
```

```
    ...
    if (c.compare(b1,b2) < 0) System.out.println("b2 más grande");
  }
```

(d) Definir y usar un comparador para `BaseDatosRel`:

```
public class CompBD implements Comparator<BaseDatosRel> {
  int compare(BaseDatosRel b1, BaseDatosRel b2) {
    if (b1.size() < b2.size()) return -1;
    else if (b1.size() == b2.size()) return 0;
         else return 1;
  }
}
public static void main(String [] s) {
  BaseDatos b1, b2;
  CompBD c = new CompBD();
  ...
  if (c.compare(b1,b2) < 0) System.out.println("b2 más grande");
}
```

(2 puntos) 4. **Se pide:** Escribir el código del método `addFirst` de la clase `NodePositionList<E>`. A continuación recordamos las primeras líneas de código de dicha clase en las que se muestran sus tres atributos:

```
public class NodePositionList<E> implements PositionList<E> {
  protected int numElts;              // Number of elements in the list
  protected DNode<E> header, trailer; // Special sentinels
  ...
  /** Inserts an element at the front of the list, creating new position. */
  public void addFirst(E element) {
    /* COMPLETAR */
  }
```

```
public void addFirst(E element) {
  DNode<E> newNode = new DNode<E>(header, header.getNext(), element);
  header.getNext().setPrev(newNode);
  header.setNext(newNode);
  numElts++;
}
```

(3 puntos) 5. **Se pide:** Implementar el método:

```
public PositionList<Integer>
 fairOrderMerge(PositionList<Integer> l1, PositionList<Integer> l2)
```

que toma como parámetros dos listas no vacías de enteros ordenados de menor a mayor, y debe devolver una nueva lista de enteros ordenados de menor a mayor, combinando los elementos de `l1` y `l2`, pero sin modificar dichas listas. Por ejemplo, supongamos que `l1` contiene los elementos 1, 3, 5, 7. Supongamos que `l2` contiene los elementos -4, 0, 2, 3, 5, 9. La lista resultado debe contener los elementos -4,0,1,2,3,3,5,5,7,9.

Solución $O(n)$ que recorre ambas listas a la vez (admite que las listas parámetro puedan ser vacías):

```
public boolean hasNext(PositionList<E> l, Position<E> p) {
  /* Método auxiliar, indica si "p" tiene un nodo siguiente en "l" */
  return p != l.last();
}
public PositionList<Integer>
fairOrderMerge(PositionList<Integer> l1, PositionList<Integer> l2) {
  NodePositionList<Integer> res = new NodePositionList<Integer>();
  Position<Integer> cur1 = l1.isEmpty() ? l1.first() : null;
  Position<Integer> cur2 = l2.isEmpty() ? l2.first() : null;

  while (cur1 != null && cur2 != null)
    if (cur1.element() < cur2.element()) {
      res.addLast(cur1.element());
      cur1 = hasNext(l1,cur1) ? l1.next(cur1) : null ;
    } else {
      res.addLast(cur2.element());
      cur2 = hasNext(l2,cur2) ? l2.next(cur2) : null ;
    }

  while (cur1 != null) {
    res.addLast(cur1.element());
    cur1 = hasNext(l1,cur1) ? l1.next(cur1) : null ;
  }

  while (cur2 != null) {
    res.addLast(cur2.element());
    cur2 = hasNext(l2,cur2) ? l2.next(cur2) : null ;
  }
  return res;
}
```

A. Código de apoyo

A.1. Interfaz `PositionList<E>`

```
package net.datastructures;
import java.util.Iterator;
/**
 * An interface for positional lists.
 * @author Roberto Tamassia, Michael Goodrich
 */
public interface PositionList<E> extends Iterable<E> {
  /** Returns the number of elements in this list. */
  public int size();

  /** Returns whether the list is empty. */
  public boolean isEmpty();

  /** Returns the first node in the list. */
  public Position<E> first();

  /** Returns the last node in the list. */
  public Position<E> last();

  /** Returns the node after a given node in the list. */
  public Position<E> next(Position<E> p)
    throws InvalidPositionException, BoundaryViolationException;

  /** Returns the node before a given node in the list. */
  public Position<E> prev(Position<E> p)
    throws InvalidPositionException, BoundaryViolationException;

  /** Inserts an element at the front of the list, creating new position. */
  public void addFirst(E e);

  /** Inserts and element at the back of the list, creating new position. */
  public void addLast(E e);

  /** Inserts an element after the given node in the list. */
  public void addAfter(Position<E> p, E e)
    throws InvalidPositionException;

  /** Inserts an element before the given node in the list. */
  public void addBefore(Position<E> p, E e)
    throws InvalidPositionException;

  /** Removes a node from the list, returning the element stored there. */
  public E remove(Position<E> p) throws InvalidPositionException;

  /** Replaces the element stored at the given node, returning old element. */
  public E set(Position<E> p, E e) throws InvalidPositionException;

  /** Returns an iterable collection of all the nodes in the list. */
  public Iterable<Position<E>> positions();

  /** Returns an iterator of all the elements in the list. */
  public Iterator<E> iterator();

}
```

A.2. Clase `DNode<E>`

```
package net.datastructures;
/**
 * A simple node class for a doubly-linked list.  Each DNode has a
 * reference to a stored element, a previous node, and a next node.
 *
 * @author Roberto Tamassia
 */
public class DNode<E> implements Position<E> {
  private DNode<E> prev, next;  // References to the nodes before and after
  private E element;    // Element stored in this position
  /** Constructor */
  public DNode(DNode<E> newPrev, DNode<E> newNext, E elem) {
    prev = newPrev;
    next = newNext;
    element = elem;
  }
  // Method from interface Position
  public E element() throws InvalidPositionException {
    if ((prev == null) && (next == null))
      throw new InvalidPositionException("Position is not in a list!");
    return element;
  }
  // Accessor methods
  public DNode<E> getNext() { return next; }
  public DNode<E> getPrev() { return prev; }
  // Update methods
  public void setNext(DNode<E> newNext) { next = newNext; }
  public void setPrev(DNode<E> newPrev) { prev = newPrev; }
  public void setElement(E newElement) { element = newElement; }
}
```